Project 2: Reinforcement Learning

Shreya S Ramanujam sramanuj@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 Small Data Set and Large Data Set

For both the small as well as the large datasets, I used standard Q-learning, where the Q-table is updated directly from observed transitions given in the dataset (s, a, r, sp) and the final policy is greedy in Q-value with respect to all the actions at a particular state. However, for the larger datasets, many states were never visited in the given offline dataset. This was making direct Q-learning unstable, since using the Q-values of unvisited states from the table can cause unwanted error propagations in the Q value estimations.

To handle this, I modified the algorithm to interpolate Q-values for unseen states using their nearest visited counterparts. Specifically, I maintained a `visited_state` list to keep track of visited states, implying their Q-values are reliable. When updating or querying the Q-table for an unvisited state s, the algorithm locates the closest visited state based on state index (between 1 and 100 for small and 1 to 302020 for large) by doing a binary search on this `visited_state` list and uses its Q-values as a proxy in the Bellman update equation. This allows smoother propagation of information across sparsely sampled regions of the state space and a better overall policy. I used this interpolation both in the Bellman update equation (for both $Q(sp)$ and $Q(s)$ where needed) and also for the final greedy policy itself. The pseudocode is given in Algorithm 1.1.

Performance Characteristics: I ran the small dataset for **10** epochs, taking **4.44 seconds**. With small, even 10 epochs were sufficient to give a good policy. I ran the large dataset for **50** epochs, taking **46.53 seconds**. For large, I needed more epochs to arrive at a reasonable policy and estimate for Q, hence the larger number of epochs.

1.2 Medium Data Set

For the Medium Data Set, I tried the same algorithm detailed in 1.1, but I was never able to pass the autograder test. I also tried to include another clause in the Q-learning algorithm: if sp is a terminal state (≥ 0.45 in the original OpenAI Gymnasium documentation, which becomes State ≥ 457 after discretization to 500 states), then the target is just r, there is no additional $\gamma \max_a Q_{target}(a)$ term. However, this was not able to match the baseline either.

Finally, I settled on a degree 5 polynomial ridge regression to fit Q values: Poly(X, y). The predictor (X) values for the regression were the normalized [position, velocity] tuples and the targets were the Bellman update targets: $y = r + \gamma \max_a Q(sp, a)$. After fitting the polynomial, we use that to fill up our Q-table and greedily select actions for every state. The pseudocode is given in Algorithm 1.2.

Performance Characteristics: I ran this algorithm for **25** epochs, which took **27.98 seconds**. Any lesser number of epochs did not give a good policy (because the Q function was not well-estimated), which is why I chose to run for 25 epochs.

Algorithm 1 Q-Learning with Nearest-Visited-State Interpolation

Initialize $Q(s, a) \leftarrow 0$ for all states s and actions a
Initialize visited states list as empty
for each epoch **do**
    for each transition (s, a, r, sp) in shuffled data **do**
        if s not visited **then**
            Mark s as visited
            $Q_{update} \leftarrow Q(\text{nearest visited state}, .)$
        else
            $Q_{update} \leftarrow Q(s, .)$
        end if
        if sp not visited **then**
            $Q(sp, \cdot) \leftarrow Q(\text{nearest visited state}, .)$
        else
            $Q_{target} \leftarrow Q(sp, .)$
        end if
        Compute target: $y \leftarrow r + \gamma \max_a Q_{target}(a)$
        Update: $Q(s, a) \leftarrow Q_{update}(a) + \alpha(y - Q_{update}(a))$
    end for
end for
for each state s **do**
    if s visited **then**
        $\pi(s) \leftarrow \arg\max_a Q(s, a)$
    else
        $\pi(s) \leftarrow \arg\max_a Q(\text{nearest visited state}, a)$
    end if
end for
return policy π

Algorithm 2 Polynomial Regression to find Q Values

Initialize $Q(s, a) \leftarrow 0$ for all states s and actions a
for each iteration **do**
    for each transition (s, a, r, s') in dataset **do**
        Compute target: $y \leftarrow r + \gamma \max_{a'} Q(s', a')$
    end for
    Fit polynomial regression $Q(s, a) \approx y$ using $[s/n_{\text{states}}, a/n_{\text{actions}}]$ as features
    Predict $Q(s, a)$ for all state-action pairs and update Q
end for
Return policy: $\pi(s) \leftarrow \arg\max_a Q(s, a)$

2. Code

```
import numpy as np
import pandas as pd
import torch.nn as nn
import argparse
import bisect
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import Ridge
from time import time

#### Helper functions, but I did not end up using them ####
def state_to_indices(position, velocity):
    pos_idx = int((position + 1.2) / (0.6 + 1.2) * 499)  # scale to [0, 499]
    vel_idx = int((velocity + 0.07) / (0.07 + 0.07) * 99)  # scale to [0, 99]
    return pos_idx, vel_idx

def indices_to_state(pos_idx, vel_idx):
    position = pos_idx / 499 * (0.6 + 1.2) - 1.2
    velocity = vel_idx / 99 * (0.07 + 0.07) - 0.07
    return position, velocity

def state_to_discretized_vals(state):
    pos_idx = (state - 1) % 500
    vel_idx = (state - 1) // 500
    return (pos_idx, vel_idx)

def nearest_visited_state_mountaincar(visited_list, s):
    pos_s , vel_s = state_to_discretized_vals(s + 1)
    best_dist = float('inf')
    nearest = None
    for v in visited_list:
        pos_v , vel_v = state_to_discretized_vals(v + 1)
        dist = abs(pos_v - pos_s) + abs(vel_v - vel_s)
        if dist < best_dist:
            best_dist = dist
            nearest = v
    return nearest

#### Unused Helper functions end ####

# function to find nearest visited neighbour for small and large
```

```python
def nearest_visited_state(visited_list, state):
    # binary search for nearest visited state
    idx = bisect.bisect_left(visited_list, state)
    if idx == 0:
        nearest = visited_list[0]
    elif idx == len(visited_list):
        nearest = visited_list[-1]
    else:
        left = visited_list[idx - 1]
        right = visited_list[idx]
        nearest = left if abs(left - state) <= abs(right - state) else right
    return nearest

# code for medium dataset algorithm
def fitted_q_iteration_poly(data, n_states, n_actions, gamma=0.99, n_iters=50, degree=5):
    model = make_pipeline(PolynomialFeatures(degree), Ridge(alpha=1.0))
    q_values = np.zeros((n_states, n_actions))

    for it in range(n_iters):
        X, y = [], []
        for _, row in data.iterrows():
            s, a, r, sp = row['s'] - 1, row['a'] - 1, row['r'], row['sp'] - 1
            target = r + gamma * np.max(q_values[sp])
            X.append([s / n_states, a / n_actions]) # to keep range same
            y.append(target)

        X, y = np.array(X), np.array(y)
        model.fit(X, y)

        sa_pairs = np.array([[s/n_states, a/n_actions] for s in range(n_states) for a in range(n_actions)])
        q_pred = model.predict(sa_pairs)
        q_values = q_pred.reshape(n_states, n_actions)
        print(f"Iteration {it+1}/{n_iters} done.")

    return np.argmax(q_values, axis=1) + 1

# code for small and large dataset algorithm
def q_learning(task_name, data, n_states, n_actions, alpha=0.1, gamma=0.95, epochs=10):
    # Initialize Q-table
    q_table = np.zeros((n_states, n_actions))  # n_states and n_actions
    # maintain a set of visited states
```

```python
    visited_states = np.zeros(n_states, dtype=bool)
    visited_list = []  # keep sorted list of visited states
    for epoch in range(epochs):
        print(f"Epoch {epoch+1}/{epochs}")
        for _, row in data.sample(frac=1, random_state=epoch).iterrows():
            s = row['s'] - 1 # since states are 1-indexed in the data
            a = row['a'] - 1 # since actions are 1-indexed in the data
            r = row['r']
            s_next = row['sp'] - 1 # since states are 1-indexed in the data
            # for q_table, find nearest state that we have visited
            # mark visited
            if not visited_states[s]:
                bisect.insort(visited_list, s)  # insert while keeping sorted
                visited_states[s] = True
                nearest = nearest_visited_state(visited_list, s)
                q_update = q_table[nearest]
            else:
                q_update = q_table[s]

            if not visited_states[s_next]:
                bisect.insort(visited_list, s_next)  # insert while keeping sorted
                nearest = nearest_visited_state(visited_list, s_next)
                q_target = q_table[nearest]
            else:
                q_target = q_table[s_next]

            # if (task_name == 'medium' and state_to_discretized_vals(s + 1)[0] >= 457): # terminal state
            #     target = r
            # else:
            target = r + gamma * np.max(q_target)

            q_table[s, a] = q_update[a] + alpha * (target - q_update[a])

    policy = np.zeros(n_states, dtype=int)

    # Derive policy using nearest visited state via binary search
    for i in range(n_states):
        if visited_states[i]:
            policy[i] = np.argmax(q_table[i])
        else:
            nearest = nearest_visited_state(visited_list, i)
            policy[i] = np.argmax(q_table[nearest])
```

```python
    return policy + 1  # convert back to 1-indexed


def main():
    # get arguments from command line with defaults
    parser = argparse.ArgumentParser()
    parser.add_argument('--task_name', type=str, default='small', help='Task name: small, medium, large')
    parser.add_argument('--alpha', type=float, default=0.1, help='Learning rate')
    parser.add_argument('--gamma', type=float, default=None, help='Discount factor')
    parser.add_argument('--num_epochs', type=int, default=None, help='Number of training epochs')
    parser.add_argument('--degree', type=int, default=None, help='Degree of polynomial features for medium task')
    args = parser.parse_args()

    if(args.task_name == 'small'):
        n_states = 100
        n_actions = 4
        input_data_path = './data/small.csv'
        output_path = 'small.policy'
        args.gamma = 0.95 if args.gamma is None else args.gamma
        args.num_epochs = 10 if args.num_epochs is None else args.num_epochs

    elif(args.task_name == 'medium'):
        n_states = 50000
        n_actions = 7
        input_data_path = './data/medium.csv'
        output_path = 'medium.policy'
        args.gamma = 1.0 if args.gamma is None else args.gamma
        args.num_epochs = 25 if args.num_epochs is None else args.num_epochs
        args.degree = 5 if args.degree is None else args.degree

    elif(args.task_name == 'large'):
        n_states = 302020
        n_actions = 9
        input_data_path = './data/large.csv'
        output_path = 'large.policy'
        args.gamma = 0.95 if args.gamma is None else args.gamma
        args.num_epochs = 50 if args.num_epochs is None else args.num_epochs

```

```python
    # read data
    data = pd.read_csv(input_data_path)
    # convert all data to ints
    data = data.astype(int)
    if args.task_name == 'medium':
        print("Using specialized Q-learning for medium task")
        start_time = time()
        policy = fitted_q_iteration_poly(data, n_states, n_actions, gamma=args.gamma, n_iters=args.num_epochs, degree=args.degree)
        print(f"Medium task completed in {time() - start_time:.2f} seconds")
    else:
        print("Using general Q-learning")
        start_time = time()
        policy = q_learning(args.task_name, data, n_states, n_actions, alpha=args.alpha, gamma=args.gamma, epochs=args.num_epochs)
        print(f"General Q-learning on {args.task_name} completed in {time() - start_time:.2f} seconds")
    # output to a file
    with open(output_path, 'w') as f:
        for _, action in enumerate(policy):
            f.write(f"{action}\n")

if __name__ == "__main__":
    main()
```